- A ***recursive function*** is a function that calls itself.

- Anything that can be solved *iteratively* can be solved *recursively* and vice versa.

- Sometimes a recursive solution can be expressed more *simply* and *succinctly* than an iterative one.

factorial Function ($n!$)

factorial(0) =	1 *(by definition)*	= 1
factorial(1) =	1*1	= 1**factorial(0)*
factorial(2) =	2*1	= 2**factorial(1)*
factorial(3) =	3*2*1	= 3**factorial(2)*
factorial(4) =	4*3*2*1	= 4**factorial(3)*
factorial(5) =	5*4*3*2*1	= 5**factorial(4)*
factorial(6) =	6*5*4*3*2*1	= 6**factorial(5)*

Recursive Definition of *factorial(n)*

$$
factorial(n) = \begin{cases} 1 & \text{if } n = 0 \\ n * factorial(n-1) & \text{if } n > 0 \end{cases}
$$

- How would we implement this in C++ ?

Function Definition:	**C++ Implementation:**
factorial(n) =	int factorial(n)
	{
	if(n == 0)
1 *if n = 0*	return 1;
	else
*n*factorial(n-1)* *if n > 0*	return n*factorial(n-1);
	}

Understanding Recursion

- You can think of a recursive function call as if it were calling a completely separate function.

- In fact, the *operations* that can be performed by both functions is the same, but the *data* input to each is different

Understanding Recursion (Cont'd.)

```
int  factorialA( int  n )
{
   if( n == 0 )
      return 1;
   else
      return  n*factorialB( n-1 );
}
```

```
int  factorialB( int  m )
{
   if( m == 0 )
      return 1;
   else
      return  m*factorialC(m-1);
}
```

- If factorialB( ) and factorialC( ) perform the same operations as factorialA( ), then factorialA( ) can be used in place of them.

Example: *factorial(3)*

factorial(3): n = 3 calls factorial(2)

factorial(2): n = 2 calls factorial(1)

factorial(1): n = 1 calls factorial(0)

factorial(0): returns 1 to factorial(1)

factorial(1): 1*factorial(0) becomes 1*1 = 1

: returns 1 to factorial(2)

factorial(2): 2*factorial(1) becomes 2*1 = 2

: returns 2 to factorial(3)

factorial(3): 3*factorial(2) becomes 3*2 = 6

: returns 6

Questions for Constructing Recursive Solutions

- **Strategy:** Can you define the original problem in terms of smaller problem(s) of the same type?
  - Example: *factorial(n) = n*factorial(n-1) for n > 0*
- **Progress:** Does each recursive call diminish the size of the problem?
- **Termination:** As the problem size diminishes, will you eventually reach a "*base case*" that has an easy (or trivial) solution?
  - Example: *factorial(0) = 1*

Example: *Slicing Sausage*

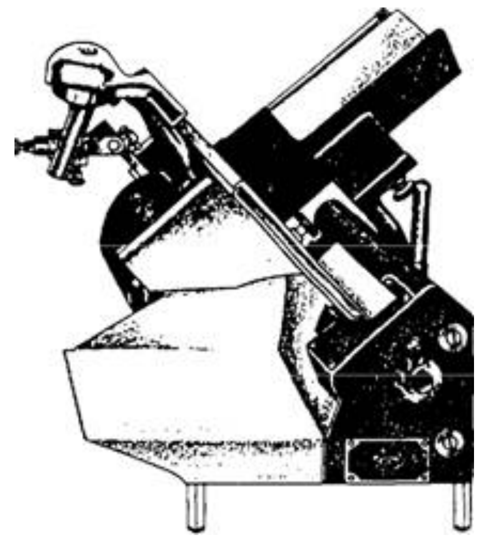
- **Problem:** Slice a sausage from back to front.
  (Assume that sausages have distinguishable front and back ends.)
- **Solution Strategy:** Slicing a sausage into N slices from back to front can be decomposed into *making a single slice at the end* (which is “easy”) and *making the remaining N-1 slices* from back to front (which is a smaller problem of the “same type”).

Slicing Sausage (Cont'd)

- **Progress:** If we keep reducing the length of the sausage to be sliced, we will eventually end up with 1 slice left.
  - We could even go a step further and end with a sausage of length 0, which requires no slicing.
- **Termination:** Since our strategy reduces the size of the sausage by 1 slice each step, we will eventually reach the *base case* (0 slices left).

Listen up! Here's the plan ...

0 1 2



Butcher #2 *always makes the first slice* at the rightmost end. He then passes the sausage to butcher #1, who makes the next cut, followed by butcher #0. They take turns with the only sausage slicer in their shop.

Sausage Slicer (in C++)

```
#define  make1slice      cout

void  sausageSlicer( char sausage[],  int size )
{
    if( size > 0 )
    {
        // slice the end off
            make1slice  <<  sausage[ size-1 ];
        // slice the rest of the sausage
            sausageSlicer( sausage, size-1 );
    }
    // base case: do nothing if size == 0
}
```

Trial Run

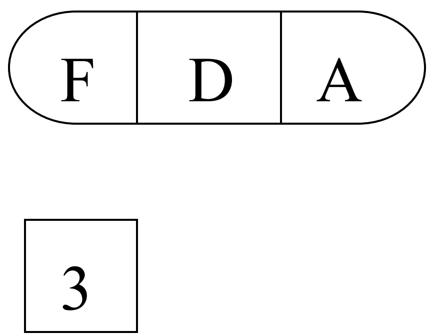
- Suppose *char pepperoni[]* contains *{'F', 'D', 'A'}*
- Executing

  sausageSlicer(pepperoni, 3);

results in

sausage

size



Trial Run (Cont'd.)

- Since size = 3 > 0,

  ```
  make1slice  <<  sausage[ size-1];
  ```

  will cause sausage[2], containing 'A', to be sliced off.

- After this

  ```
  sausageSlicer( sausage, 2 );
  ```

  is executed.

Trial Run (Cont'd.)

- Executing

  sausageSlicer(sausage, 2);

  causes

  make1slice << sausage[size-1];

  to be executed, which results in sausage[1], containing 'D', to be sliced off.

- After this

  sausageSlicer(sausage, 1);

  is executed.

Trial Run (Cont'd.)

- Executing

  sausageSlicer(sausage, 1);

  causes

  make1slice << sausage[size-1];

  to be executed, which results in sausage[0], containing 'F', to be sliced off.
- After this

  sausageSlicer(sausage, 0);

  is executed.

Trial Run (Cont'd.)

- Executing

  sausageSlicer(sausage, 0);

  does *nothing* and returns to the place where it was called.

Trial Run - *Return Path*

- sausageSlicer(sausage, 0) returns to sausageSlicer(sausage, 1), which has nothing left to do.
- sausageSlicer(sausage, 1) returns to sausageSlicer(sausage, 2), which has nothing left to do.
- sausageSlicer(sausage, 2) returns to sausageSlicer(sausage, 3), which has nothing left to do.
- sausageSlicer(sausage, 3) returns to sausageSlicer(pepperoni, 3), the original call to sausageSlicer( ), and execution is done.

Trial Run - *Key Point*

Note that there is ***only one*** *sausageSlicer,* (i.e. one recursive function), but it is used over and over on successively smaller pieces of the original sausage until, finally, the entire sausage is sliced.

New Strategy for a *New Tool*

- **Solution Strategy:** Slicing a sausage into N slices from back to front can be decomposed into *slicing N-1 slices from back to front* (a smaller problem of the same type) and *making a single slice at the front* (which is "easy").
- **Progress & Termination:** Since, as before, our strategy reduces the size of the sausage by 1 slice each step, we will eventually reach the *base case* (0 slices left).

New Tool … New Strategy

0 1 2

This time, someone hands the sausage to butcher #0. As the senior member of the team, he will slice only if the others have done their work. So, he passes the sausage to butcher #1 who, in turn, passes the sausage to butcher #2. Butcher #2 makes the first slice, as before, at the rightmost end of the sausage, and then passes it back to the other two butchers, who can now complete their tasks.

New Sausage Slicer in C++

```
int  size;            // global variable containing size of sausage

void  sliceAsausage( char sausage[],  int pos )
{
   if( pos < size )
   {  // cut into slices everything to the right of sausage[ pos ]
         sliceAsausage( sausage, pos+1 );
      // slice off sausage[ pos ];
         make1slice  <<  sausage[ pos ];
   }
   // base case: do nothing if pos == size (i.e. past end of sausage)
}
```

Trial Run of New Sausage Slicer

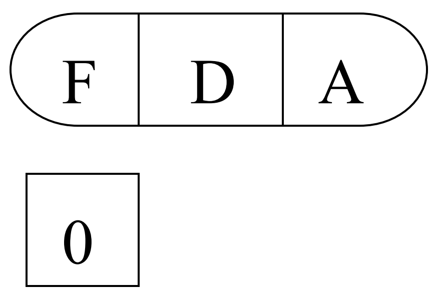
- Suppose, as before, *char pepperoni[]* contains *{‘F’, ‘D’, ‘A’}* **and** *size* is initialized to 3.
- Executing

  sliceAsausage(pepperoni, 0);

  results in

  sausage

  pos



New Slicer Trial Run (Cont'd.)

- Since pos = 0 < size,

  ```
  sliceAsausage( sausage, 1 );
  ```

  will be executed.
- After this

  ```
  sliceAsausage( sausage, 2 );
  ```

  is executed, followed by

  ```
  sliceAsausage( sausage, 3 );
  ```

New Slicer Trial Run - Return Path

- sliceAsausage(sausage, 3) does nothing since pos = size.
- sliceAsausage(sausage, 3) returns to sliceAsausage(sausage, 2), which prints sausage[2] = ‘A’.
- sliceAsausage(sausage, 2) returns to sliceAsausage(sausage, 1), which prints sausage[1] = ‘D’.
- sliceAsausage(sausage, 1) returns to sliceAsausage(sausage, 0), which prints sausage[0] = ‘F’.
- sliceAsausage(sausage, 0) returns to sliceAsausage(pepperoni, 0), and execution is done.

There’s more than one way to slice a sausage!

X^n Function

$$X^n = 1 \qquad \text{if } n = 0 \text{ (base case)}$$
$$X^n = X*X^{(n-1)} \qquad \text{if } n > 0$$

This can easily be translated into C++. However, a *more efficient* definition is possible:

$$X^n = 1 \qquad \text{if } n = 0 \text{ (base case)}$$
$$X^n = [X^{(n/2)}]^2 \qquad \text{if } n > 0 \text{ and even}$$
$$X^n = X*[X^{(n-1)/2}]^2 \qquad \text{if } n > 0 \text{ and odd}$$

C++ Implementation of X^n

```
double  power( double X,  int n )
{                                   // Note: Iterative solution is more efficient
      double  HalfPower;
      if( n == 0 ) return 1;
      if( n % 2 == 0 )      // n is even
      {
            Halfpower = power( X, n/2 );
            return  HalfPower*HalfPower;
      }
                            // n is odd
      Halfpower = power( X, (n-1)/2 );
      return  X*HalfPower*HalfPower;
}
```

Fibonacci Sequence

The first two terms of the sequence are 1, and each succeeding term is the sum of the previous pair.

1 1

1 + 1 = 2

1 + 2 = 3

2 + 3 = 5

3 + 5 = 8

5 + 8 = 13 . . . , or

1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 . . .

Fibonacci Sequence (Cont'd.)

Function Definition:	**C++ Implementation:**
	`int fib( int n )`
	`{`
fib(1) = 1 (base case)	`if( n <= 2 )`
fib(2) = 1 (base case)	`return 1;`
	`else`
fib(n) = fib(n-1) + fib(n-2), for n > 2	`return fib(n-1) + fib(n-2);`
	`}`

- 2 base cases and 2 simpler problems of the "same kind"
- Very inefficient: fib(7) will call fib(3) five times!

Fibonacci Sequence with Rabbits

- Problem posed by Fibonacci in 1202:
  - A pair of rabbits 1 month old are too young to reproduce.
  - Suppose that in their 2^{nd} month and every month thereafter they produce a new pair.
  - If each new pair of rabbits does the same, and none of them die, how many pairs of rabbits will there be at the beginning of each month?

Fibonacci Sequence with Rabbits (Cont'd.)

Month 1:	# Pairs:	1	Adam & Eve
2:		1	Adam & Eve
3:		2	Adam & Eve have twins1
4:		3	Adam & Eve have twins2
5:		5	Adam & Eve have twins3; twins1 have twins4
6:		8	Adam & Eve have twins5; twins1 have twins6; twins2 have twins7

- *Result: #pairs follows the Fibonacci sequence!*

Fibonacci Sequence - Other Applications

- A male bee has only one parent (his mother), while a female bee has a father and a mother. The number of ancestors, per generation, of a male bee follows the Fibonacci sequence.
- The number of petals of many flowers are Fibonacci numbers.
- The number of leaves at a given height off the ground of many plants are Fibonacci numbers.

Mad Scientist's Problem

A mad scientist wants to make a straight chain of length n out of pieces of lead and plutonium. However, the mad scientist is *no dummy!* He knows that if he puts two pieces of plutonium next to each other, the whole chain will explode. How many safe, linear chains are there?

Example: n = 3

L L L (safe)	P L L (safe)
L L P (safe)	P L P (safe)
L P L (safe)	P P L (unsafe)
L P P (unsafe)	P P P (unsafe)

Result: 5 safe chains

Mad Scientist (Cont'd.)

Let $C(n)$ = number of safe chains of length n

$L(n)$ = number of safe chains of length n *ending with lead*

$P(n)$ = number of chains of length n *ending with plutonium*

Now, the total number of safe chains of length n must be the sum of those that end with lead and those that end with plutonium, namely

$$C(n) = L(n) + P(n)$$

Mad Scientist (Cont'd.)

Note that we make a chain of length n by adding to a chain of length n-1.

So, consider all chains of length n-1. Note that we can add a piece of lead to the end of each of these, since this will not make the chain unsafe.

Therefore,

$$L(n) = C(n-1)$$

Mad Scientist (Cont'd.)

Consider again all chains of length n-1. Note that we can add a piece of plutonium to the end of only the chains that end with lead.

Therefore,

$$P(n) = L(n-1)$$

Mad Scientist (Cont'd.)

Substituting formulas for L(n) and P(n) in the formula for C(n) we see that

$$
\begin{aligned}
C(n) &= L(n) + P(n) \\
&= C(n-1) + L(n-1) \\
&= C(n-1) + C(n-2), \quad \text{since } L(k) = C(k-1) \text{ for any } k
\end{aligned}
$$

Note that this is the Fibonacci recursion!

However, the base case(s) are different:

$C(1) = 2$ L or P

$C(2) = 3$ LL or LP or PL

Mad Scientist (Cont'd.)

Back to our example with n = 3:

$$
\begin{aligned}
C(3) &= C(2) + C(1) \\
&= 3 + 2 \\
&= 5
\end{aligned}
$$

which agrees with the answer we found by enumerating all the possibilities.

Mr. Spock’s Dilemma

There are n planets in an unexplored planetary system, but there is only time (or fuel) for k visits.

How many ways are there for choosing a group of planets to visit?

Let C(n, k) denote the number of ways to choose k planets from among n candidates.

Mr. Spock's Dilemma: *Solution Strategy*

Consider planet Vega. Either we visit Vega or we don't.

- *If we visit* Vega, then we will have to choose k-1 other planets to visit from the remaining n-1.
- *If we don't visit* Vega, then we will have to choose k other planets to visit from the remaining n-1.
- Therefore,

$$C(n, k) = C(n-1, k-1) + C(n-1, k) \quad \text{for } 0 < k < n$$

Mr. Spock's Dilemma: *Recursion Criteria*

Consider the criteria for constructing a recursive solution:

1) **Strategy:** Is the original problem defined in terms of smaller problems of the same type? *Yes,*

$$C(n, k) = C(n-1, k-1) + C(n-1, k)$$

2) **Progress:** Does each recursive call diminish the size of the problem? *Yes, first argument of C decreases with each recursive call and second argument does not increase.*

3) **Termination:** Will a "base case" be reached eventually? *Let's see what base cases are needed, and then see if one of them will always be reached.*

Mr. Spock's Dilemma: *Base Cases*

- Note that the recursion formula

  $$C(n, k) = C(n-1, k-1) + C(n-1, k)$$

  only applies when $0 < k < n$. Consequently, we need to consider $k < 0$, $k = 0$, $k = n$, and $k > n$.
- Since there is only 1 way to choose 0 planets and only 1 way to choose all n planets, we have

  $$C(n, k) = 1 \quad \text{if } k = 0 \text{ or } k = n$$

- Since it is not possible to choose < 0 planets or > n planets,

  $$C(n, k) = 0 \quad \text{if } k < 0 \text{ or } k > n$$

Base Cases (Cont'd.)

- Putting this all together, we have

  $$C(n, k) = \begin{cases} 0 & \text{if } k < 0 \text{ or } k > n \quad \textit{(base case)} \\ 1 & \text{if } k = 0 \text{ or } k = n \quad \textit{(base case)} \\ C(n-1, k-1) + C(n-1, k) & \text{if } 0 < k < n \end{cases}$$

- Consider the recursion formula, where $0 < k < n$. Since the first argument of $C(n, k)$ decreases with each recursive call and second argument does not increase, eventually either $n = k$ or $k = 0$. Both *base cases* are defined above. Therefore, *termination is assured.*

Mr. Spock's Dilemma: *Solution in C++*

```
int  C( int n,  int k )      // # of ways to choose k of n things
{
      if( k == 0 || k == n ) return 1;
      if( k  <  0 || k  >  n ) return 0;

      return C( n-1, k-1 ) + C( n-1, k );
}
```

Binary Search: *Telephone Book*

- Problem: Search the telephone book for someone's phone number.
- Binary Search Strategy:

a) Open the book somewhere near the middle.

b) If the the person's name is in the first half, ignore the second half, and search the first half, starting again at step a).

c) If the the person's name is in the second half, ignore the first half, and search the second half, starting again at step a).

d) If the person's name is on a given page, scan the page for the person's name, and find the phone number associated with it.

Binary Search: *Search an Array*

- Problem: Given an array, A[ ], of n integers, sorted from smallest to largest, determine whether value v is in the array.
- Binary Search Strategy:

  If n = 1 then check whether A[0] = v. Done.

  Otherwise, find the midpoint of A[ ].

  If v > A[midpoint] then recursively search the second half of A[ ].

  If v <= A[midpoint] then recursively search the first half of A[ ].

Search an Array: C++ Implementation

```
int  binarySearch( int A[ ], int v, int first, int last )
{
   if( first > last ) return -1;                // v not found in A[ ]
   int  mid = (first + last)/2;                 // set mid to midpoint
   if( v == A[mid] ) return mid;
   if( v  <  A[mid] ) return binarySearch( A, v, first, mid-1 );
   return binarySearch( A, v, mid+1, last );
}
```

C++ Implementation (Cont'd.)

Two common mistakes:

1) CORRECT: `mid = ( first + last )/2;`
   INCORRECT: `mid = ( A[first] + A[last] )/2;`

2) CORRECT: `return binarySearch( A, v, mid+1, last );`
   INCORRECT: `return binarySearch( A, v, mid, last );`

Search an Array: Implementation Notes

- The whole array, A[ ], is passed with each call to binarySearch( ).

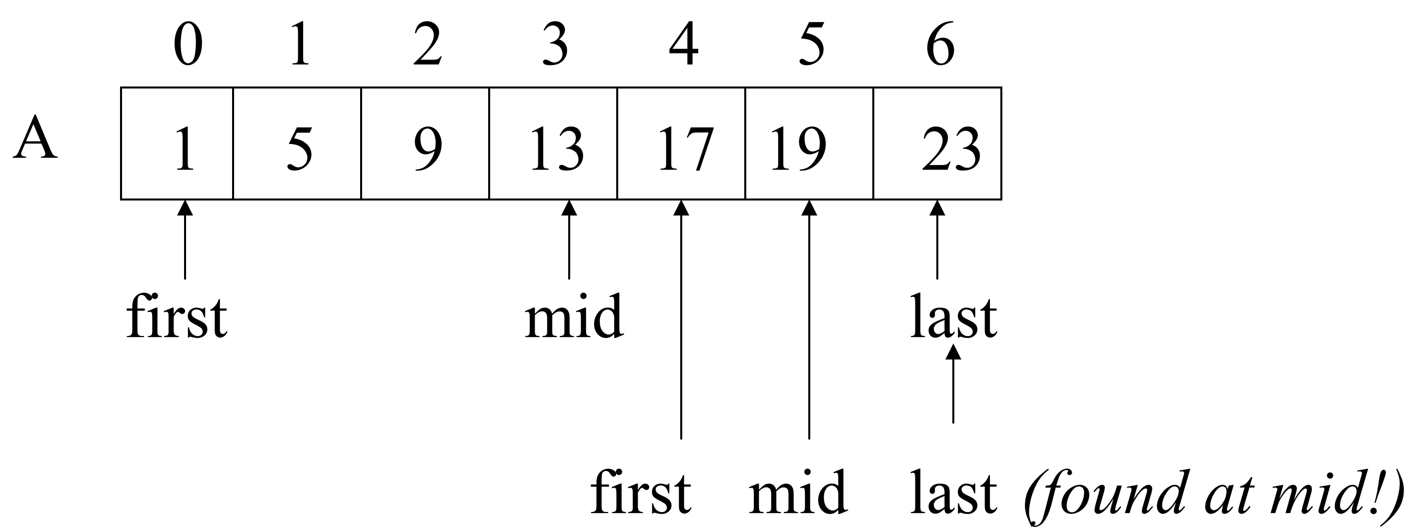
- The active part of array A[ ] is defined by *first* and *last*.

- A return value of -1 means that *v* was not found.

Search an Array: Example

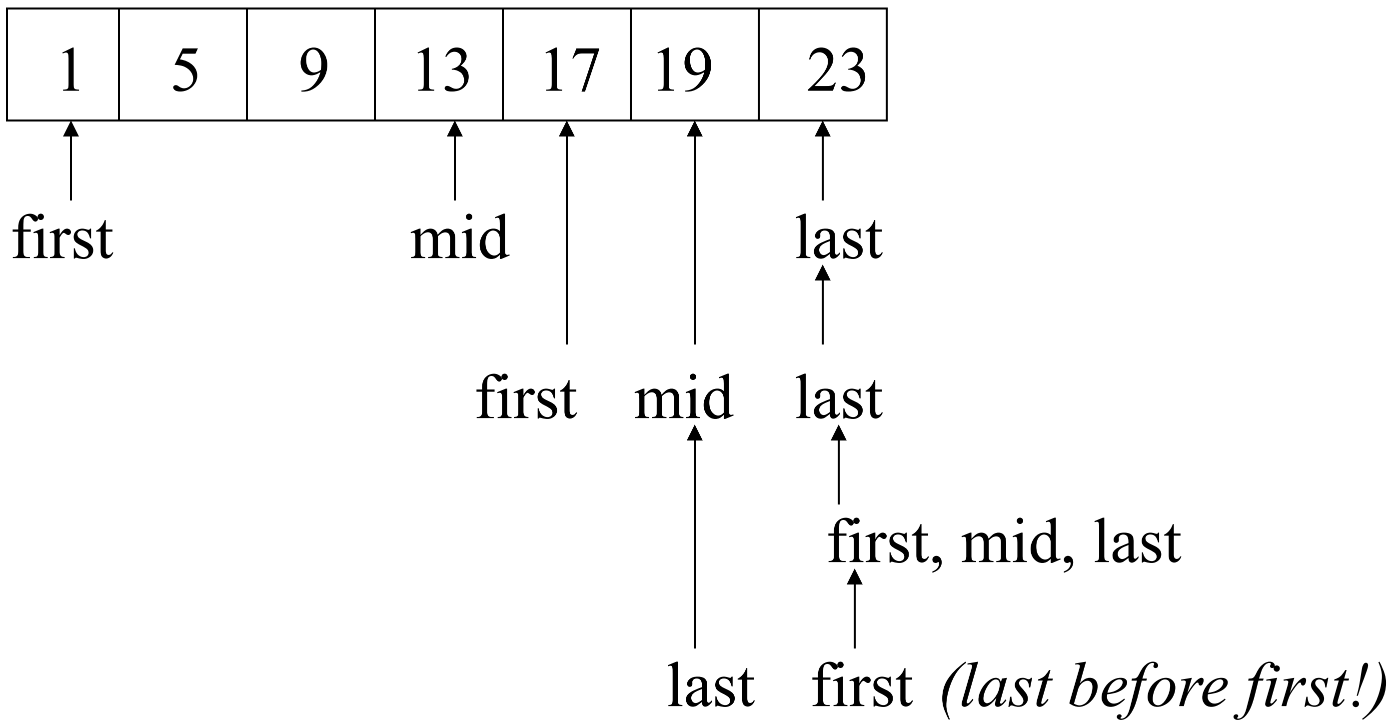
- Suppose *int A[ ]* contains *{1, 5, 9, 13, 17, 19, 23}*, and we are interested in searching for ***19***.
- Executing *binarySearch(A, **19**, 0, 6);*

results in



Search an Array: Example (Cont'd.)

- Suppose we are interested in searching for **21:**



Search an Array: Final Comments

- Suppose that we have an array of a million numbers.
- The first decision of a binary search will eliminate approximately half of them, or 500,000 numbers.
- The second decision will eliminate another 250,000.
- Only 20 decisions are needed to determine whether a given number is among a sorted list of 1 million numbers!
- A *sequential* search might have to examine *all of them*.
- Additional Note: Binary searching through a *billion* numbers would require about 30 decisions, and a *trillion* numbers would (theoretically) require only 40 decisions.